The Zipper

Henri-Charles Blondeel
Pablo Pera

February 22, 2007

Contents

1 Introduction

There exist many applications that make use of destructive operations over tree-like data structures. When the amount of data grows, performing a copy of it becomes a serious drawback to their efficiency. *The Zipper*, introduced by Gérard Huet in [1], is an interesting approach that overcomes this inconvenient.

The basic idea is to separate the part of the tree that is really in use, from the rest of it. This way, operators are applied over a small subtree, while most of the data keeps unaccessed unless necessary.

The other great advantage of this kind of data structures is the navigation ability. Some navigation primitives are provided, to be able to change our attention focus in a fast, painless way. They allow movement in the four 2D directions: up, down, left and right.

To accomplish this, Huet proposed a bottom-to-top approach: from the focus of attention, we keep record of all the siblings[1] for each step in height, until we eventually reach the top. Form this information, together with the focus attention subtree, one can easily return to the classical tree representation. The navigation primitives are defined accordingly.

Nonetheless, other approaches are also valid. Taking advantage of polymorphism allows to design higher order implementations of the same idea, so the further abstraction level gives more flexibility to the user of the zipper. The underlying principles for constructing such a structure are presented by Mark P. Jones in [2], the base of the second implementation we will present later.

Conor McBride reminds us in [3] that this problem can be handled from other points of view. If one understands the focus of attention as a hole, and the rest of the tree as its context, some other relevant techniques can be applied, such as a kind of derivatives of regular types. Thus, applying the

[1] Which are trees themselves.

notion of partial differentiation, one-hole contexts are given for elements of recursive types, making a surprising relation with Huet's original zipper.

We will also try to contribute with our own zipper representation, which will be presented in the subsection 2.4 on page 8.

2 Implementation

2.1 Huet's original Zipper

First, we implemented the Huet's version of a zipper over a Rose Tree. For this part, we only needed to understand how the zipper works, to then write it into Haskell.

We chose to work with a Rose Tree as well, since it represents a tree with an arbitrary number of subnodes for each parent node. We need to write the definition for a tree, and also for a path. The former has the classical tree definition, but the latter can be more tricky to interpret: it contains all the tree except a subtree, from which goes up to the root, keeping all the nodes (thus, all the subtrees) that encounters by its ascending way. Their definitions are these:

```
data Tree a = Leaf a | Node [Tree a]
```

```
data Path a = Top | MkPath [Tree a] (Path a) [Tree a]
```

Therefore, it is easy to guess that the whole tree is composed of two parts: the subtree[2], and the rest of the tree[3]. They are stored together thanks to the `Loc` data type, that has the following definition:

```
data Loc a = MkLoc (Tree a) (Path a)
```

Before getting into the navigation issue, we can ask this question: Does this new representation support the standard toolbox often used in functional programming with trees? Will we be able to define a map and a fold function for a location?

[2]Represented as a Tree, it constitutes our focus of attention
[3]Contained in the Path data type.

Mapping is not a problem, since we are able to map a function over recursive datatypes or any list of elements. We easily defined a map function for `Tree`, and then for a `Path`. And it is immediate to notice that mapping f over a `Loc` consists of mapping f over them both. Check its code in Figure 1.

Figure 1: `map` and `fold` primitives for the basic zipper.

```
mapTree ::  (a -> b) -> Tree a -> Tree b
mapTree f (Leaf a) = Leaf (f a)
mapTree f (Node (x:xs)) =
           Node ((mapTree f x):(map (mapTree f) xs))

mapPath ::  (a -> b) -> Path a -> Path b
mapPath f Top = Top
mapPath f (MkPath tree_left pathup tree_right) =
MkPath m_l m_u m_r
    where
        m_l = map (mapTree f) tree_left
        m_u = mapPath f pathup
        m_r = map (mapTree f) tree_right

mapLoc ::  (a -> b) -> Loc a -> Loc b
mapLoc f (MkLoc tree path) = MkLoc (mapTree f tree)
                                   (mapPath f path)

foldlTree ::  (b -> b -> b) -> (a -> b) -> b -> (Tree a) -> b
foldlTree g f e (Leaf a) = g e (f a)
foldlTree g f e (Node []) = e
foldlTree g f e (Node (x:xs)) = foldlTree g f
                                (foldlTree g f e x) (Node xs)
```

Regarding to the fold function, the user must know if he wants to fold an operator over the whole zipper or only over the focus. Folding over the focus is quite easy. We only need to choose a policy, such as depth-first and associating to the left, then a fold function for a roseTree can be found in any good Haskell tutorial.

But if one wants to fold over the whole tree, then the structure of the zipper

is particulary inadequate. It is not possible to respect a given order without first bringing back the focus to the node. This is due to the structure of the path which holds all the uncles of a tree. So if we want to recursively apply fold on the path we will fold over younger and older uncles before or after folding over the father... which is not the order one may want for a fold function. Our implementation is shown in the Figure 1.

Let us now get into the navigation issue. We want to be able to change the attention focus by moving to the left, right, up or down. For any of them, the path will need to be updated. Of course some moves might not be available in some cases[4]. We chose to return a `Maybe Loc a` instead of a `Loc a` to denote this failure case.

The immediate issue that arose from this decision is the fact that, after a move, a `Loc` turns into a `Maybe Loc`, which is not a valid pattern. Fortunately, the bind operator of the Maybe Monad successfully deals with this unwanted situation.

The way the navigation primitives is defined is quite intuitive: they switch subtrees between the focus of attention and the context. The only one that does not need any information from the context is the `goDown` function, shown in the Figure 2. It just takes the eldest son of the current focus of attention as the new one, and adds its siblings to the path.

Figure 2: Implementation of `goDown`.

```
goDown ::  Loc a -> Maybe (Loc a)
goDown (MkLoc (Leaf a) path) = Nothing
goDown (MkLoc (Node []) path) = Nothing
goDown (MkLoc (Node (t:ts)) path) = Just (MkLoc t (MkPath []
        path ts))
```

However, the rest of them need to get some information from the context, to be able to add it to the hole. We have an example in Figure 3 on the following page, where the Haskell code for the `goRight` primitive is revealed. It takes

[4]For instance, moving upwards from the top.

the immediate sibling in its right as the new focus of attention, giving the former focus back to the context. The rest of the navigation functions have a similar underlying reasoning.

Figure 3: Implementation of goRight.

```
goRight ::  Loc a -> Maybe (Loc a)
goRight (MkLoc t Top) = Nothing
goRight (MkLoc t (MkPath left up (head:right)))
        = Just (MkLoc (head) (MkPath (t:left) up right))
goRight (MkLoc t (MkPath left up [])) = Nothing
```

2.2 Towards more general datatypes

Reading Jones's paper, [2], we learnt it was possible to represent a recursive datatype as the fix point of a Functor. For instance, if we want to redefine `data ListInt = Nil | Cons Int ListInt`, we can do this using the pattern functor `data ListF a b = Nil | Cons a b` and the following data structure[5]: `data Mu f a = In (f a (Mu f a))`.

Then, we get a list parameterized by the type parameter `a` with type `List a = Mu ListF a`. According to this, `List a` is the fix point of `ListF a`. Then `Mu ListF Int` is a list of integers.

By using types defined by Mu one can write much more general functions.

2.3 Contexts and Derivational class

One may want to implement the zipper over other data structures.

Depending on the data structure, one will be obliged to define the update operations for the focus and the context using the constructor's data type. So one part of this task depends on the implementation of the data structure itself.

[5] Fix point of the functor *(f a)*

A second part depends on the spatial representation that one has of his data structure. Indeed, what does it mean to go to the left, or to go up? We are so used to the spatial representations we usually give to common data structures that we forget that their structure exists independently. In order to navigate over a data-type, one must bind moves to the application of the constructors.

But once this is done, the last part, defining the navigation primitives, is the same for every data type. If a move corresponds to going further inside the attention focus, then we will just have to add some information to the context. The other possibilities are that either we can not move in a given direction, or we need to retrieve a piece of the context and merge it to the former one to get our new attention focus.

Figure 4: Derivational toolbox.

```
-- canStep:  is it allowed to step in this direction?
canStep ::  (Mu f a, [Mu g b]) -> Direction -> Bool

-- step:  it extracts the new info to add to the context
step ::  (Mu f a, [Mu g b]) -> Direction -> Mu g b

-- give :  the new attention focus within the former one
give ::  (Mu f a, [Mu g b]) -> Direction -> Mu f a

-- canMerge:  does this move correspond to a merge?
canMerge ::  (Mu f a, [Mu g b]) -> Direction -> Bool

-- merge:  merge the former attention focus with the context
merge ::  (Mu f a, [Mu g b]) -> Direction -> Mu f a

-- updateContext:  the new context after performing a merge
updateContext ::  (Mu f a, [Mu g b]) -> Direction -> [Mu g b]
```

The project instructions demand to create a class Derivational which should instanciate datatypes candidates to the navigation. This gives two important advantages:

- We can use a simple algorithm using only the methods of this class to define the navigation primitives. This way this algorithm will be valid for each instance datatype
- By writing the class' methods for a certain data type, one will have to give the information about the data type and his spatial representation mentioned before. This way the task gets simplified.

Let us wonder now which methods should the class Derivational ask for. We want to be able to do all possible navigation movements using its methods in a unique algorithm.

As we have seen, a move is either a step inside the focus or a merge with the context. So, we finally come up with the toolbox in Figure 4.

Figure 5: `go` function for navigation over Derivational.

```
go (focus, listSteps) dir = if (canStep zip dir)
    then (give zip dir, ((step zip dir) :  listSteps))
    else if (canMerge zip dir)
        then ((merge zip dir), (updateContext zip dir))
        else zip
            where zip = (focus, listSteps)
```

Now, assuming that these methods are well implemented, we can blindly write the `go` method of a class `Navigation` for any instance as shown in the Figure 5.

2.4 A simple up-to-bottom approach

We have also come up with an interesting way of keeping a hole and its context in a data structure, regardless of the actual kind of tree representation that one chooses to use.

It is based on the assumption that the user[6] will write their own tree, and

[6]By user we mean the programmer that uses our zipper implementation for their own tree-like structures.

will also provide a function that returns, for any node, a list with its subtrees. This should be possible for any non-connected graph, that is, for any tree.

Our zipper consists of the basic navigation primitives, which have the same meaning as in the original zipper, but a very different implementation.

First, our path is understood as going from the root to the focus of attention. Since the number of children for every node is unknown, we will use integers to store the *decision* we make in every step in order to reach the hole, that is, a n will mean that we choose the *n-th* son to get closer to the focus of attention. For each of these steps, we will store the rest of the sons we pass by, which are trees themselves. We have named each of these steps as a `PathElement`, so the whole context is a list of steps, kept in the `Path` data type:

```
type PathElement a = (Int, [Tree a])
```

```
data Path a = Path [PathElement a]
```

A priori, we do not know the exact structure of the tree, but we do know that our focus of attention is a user-defined tree. And, as mentioned above, we will assume that a function `giveSubtrees` has been defined[7]. So we define the following data type to store all the zipper structure:

```
data DownTree a b = DownTree (Path a, Tree b)
```

The first navigation function that one should use is `goDown`. It is the only one that needs to know the subtrees of the current node. The first of them will be stored as the focus of attention, and an element will be appended to the path list containing a one (indicating that we navigate to the eldest son), and the rest of the subtrees as its second component.

The `goLeft` and `goRight` primitives are quite similar. They take the last path element, increment or decrement the decision integer, and switch between the current focus of attention and the immediate elder or younger sibling, respectively.

[7] It will have the heading `giveSubtrees :: Tree a -> [Tree a]`.

Last, the `goUp` primitive needs to gather the information contained in the focus of attention, with what the last element of the path contains, to then expand the focus of attention tree.

3 Efficiency

3.1 Huet's implementation

Space complexity

It is important to note that there is no redundant information in Huet's zipper. Therefore, if n is the number of elements in the whole tree, the space complexity is linear with it, that is, *O(n)*.

Time efficiency

First let us consider a `goDown` operation. With the basic solution, the time required for this operation is the time needed to get the focus subtree, which is constant, *O(1)*. With Huet solution we also have to update the context[8]. Actually there are two lists, one with the elder siblings and one with the younger ones, but as we access the eldest son, the first is empty and the second is the tail of the former node. So this takes constant time too.

To make `goLeft` and `goRight` faster, the list of older siblings is kept reversed, that means that the first element is always the youngest. This way, `goLeft` and `goRight` only need to access the first element of a list, which of course takes constant time.

But this last trick has a drawback. When we execute a `goUp`, the list of elder siblings must be reversed before being concatenated with the list of younger siblings, so it takes *O(n)*, being n the number of siblings of the node.

[8] That is, append a path with the list of siblings to the already existing path.

3.2 Up-to-bottom implementation

Space complexity

There is not much to say regarding space complexity, since no extra information is stored apart from the integers that represent the path from the root to the focus of attention. If n is the number of nodes, there are *log(n)* integers, that don't change the asymptotic cost, that is linear with n, *O(n)*.

Time complexity[9]

The functions `goUp`, `goRight` and `goLeft` can be analyzed together, given that they share and combine in the same way the same functions, with slight variations that do not affect the time complexity[10]. Besides, these subfunctions are not nested, so we will only need to identify the most time consuming.

Two of them depend linearly on the number of elements of the path, namely `dropLast` and `lastDecision`. We have already observed the the number of elements of the path is logarithmic with respect to the number of nodes.

The remaining are `elderSiblings` and `youngerSiblings`, that make use of linear Haskell operators on the number of siblings of a given node. This means that their worst case is linear with the number of nodes of the whole tree.

However, the function `goDown` only makes use of constant operators. But before returning, it will need to call to the user-defined function `giveSubtrees`, that will probably have constant or linear time complexity.

Therefore we can conclude that `goUp`, `goRight` and goLeft are *O(n)*, while `goDown` is *O(giveSubtrees)*, being n the number of nodes of the tree, and

[9] The source code is located at the appendix A.

[10] We will not demonstrate it, but it is immediate to see the analogy between, for example `take (lastDecision-1) lastTreeList` and `take (lastDecision-2) lastTreeList`, which appear in `goRight` and `goLeft`, respectively.

the cost of `giveSubtrees` linear in the case of a typical rose tree, or constant for a binary tree.

4 Discussion

The very first problem we encountered was Jones' paper itself. Understanding the higher-order polymorphism background he proposed is very tricky at the beginning, since their principles are quite theoretical, and also their current Haskell implementation leads to recursive types errors which are not trivial to correct.

Though the Huet's zipper proposal is simple and intuitive, McBride's paper about derivatives of regular types introduces much new syntax and abstract concepts about data-types, and is not easy to relate to our functional programming knowledge at a first sight.

When dealing with the implementation of the derivational approach, we got some trouble due to the fact that the classes `Derivative` and `Navigation` have several type parameters (not only one, which constitutes the usual example). The Haskell compiler often refuses the use of class methods in this case, seemingly to prevent type ambiguity.

In principle, we wanted that the methods of Derivational took only the arguments they needed[11]. But giving the whole zipper as an argument to each of these functions made the compiler accept to use methods defined in the class instantiation, so we adopted this form for all class methods.

5 Questions

Question 1

Why are `+` and `*` sometimes used for the datatypes `data Either a b = Left a | Right b` and `data Pair a b = Pair (a,b)` ?

[11] For instance, there is no need of knowing the context for a function that only steps further into the focus of attention.

The data type `Either` represents merely a choice, whereas `Pair` represents a composition. Thus, by using `+` and `*`, instead of `Either` and `Pair`, we get an algebraical notation.

Then, McBride observed that the type of the one-hole contexts of a recursive type built using `+` and `*` can be *computed* using algebraical formula, making this notation useful.

Question 2

Give a couple of examples of derivatives and discuss the relationships to first-year calculus.

We can take McBride's example with binary trees: `btree = 1 + btree`^{2}. Then the derivative is `btree' = 2 * btree`, where `2` has type `Bool` since it represents a choice between two possibilities. We notice that basic calculus gives us the same result: `(1 + btree`^{2}`)' = 2 * btree`.

We can take a list as another example: `list = 1 + list`. First-year calculus says `(1+list)' = 1`. Then we can ensure that the type of the derivative of a list of a is `a`.

Let us now take a more complex data structure, such as `rTree = 1 * list rTree`. It is not obvious what one may take for the type of the context, but applying the formula we get `rTree' = 1 * rTree`. That is to say; for a Rose Tree of `a`, the corresponding type is a pair of an a and a Rose Tree. This seems correct since either we navigate past an `a` and we need to keep trace of it, or we navigate in the list of `rTree` and then we leave another `rTree` behind.

Question 3

Argue that your implementation is more general than that of G. Huet.

Huet's zipper can be used only with a RoseTree because he is dealing directly with the constructors of this datatype.

By doing an instance declaration of class Derivational, the user indicates in which direction their data structure can grow. Therefore, we can, for example, consider a list which grows to the right[12] or a tree which grows downwards[13].

Question 4

There is a problem with the implementation of the type class Derivational since the derivative's type depends on the actual data type. How did you get around this problem?

The way we did it first was to give the type for the context as a class parameter. This is not supported by GHC unless we allow it using the corresponding flag, which leads to trouble when trying to make instance of this class.

6 Organization

For most of the stages of the project, we both have been working together in the lab computers. Due to the lack of time of the last days, we decided to work separately. Anyway we have always been working at the same time, either talking in person (in the lab room) or emailing each other frequently (at home), so we consider the team work has been quite efficient.

[12] i.e. the recursive element is appended to the left.
[13] i.e the subtree is given by `goUp`.

References

[1] Gérard Huet, «The Zipper (Functional Pearl)», *Journal of Functional Programming* 7 (5) 549-554, September 1997.

[2] Mark P. Jones, «Functional Programming with Overloading and Higher-Order Polymorphism», *LNCS* 925, Springer Verlag, 1995.

[3] Conor McBride, «The Derivative of a Regular Type is its Type of One-Hole Contexts», Unpublished manuscript, 2001.